NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

PETRI NET AND FAULT TREE ANALYSIS:
COMBINING TWO TECHNIQUES FOR A SOFTWARE SAFETY
ANALYSIS ON AN EMBEDDED MILITARY APPLICATION

by

Richard J. McGraw, Jr.

December, 1989

Thesis Advisor: Timothy J. Shimeall

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved For Public Release; Distribution Is Unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Naval Postgraduate School — 6b. OFFICE SYMBOL (If applicable): 37	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION — 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
PETRI NET AND FAULT TREE ANALYSIS: COMBINING TWO TECHNIQUES FOR A SOFTWARE SAFETY ANALYSIS ON AN EMBEDDED MILITARY APPLICATION (U)

12. PERSONAL AUTHOR(S)
MCGRAW, JR., RICHARD J.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM ____ TO ____	December 1989	63

16. SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Petri Nets; Fault Tree

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Software safety ensures that software programs perform within certain limits without resulting in an unacceptable risk of an accident occurring. Petri Nets are used to model concurrent computer systems by helping to understand complex interactions and paths of execution. Fault Tree analysis is used to determine safety requirements by detecting software logic errors. They also identify multiple failure sequences in a system that can lead to safety hazards. Petri Nets and Fault Tree analysis can be combined and used in conjunction with each other. They can take advantage of each others inherent strengths. This combined methodology can provide an efficient and effective safety analysis technique.

This thesis surveys software safety research and focuses on Petri Nets and Fault Tree analysis. We discuss an extended example combining Petri Nets and Fault Tree analysis. The example is a real-time, military embedded software application. We then indicate directions for further research.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL: Professor Timothy J. Shimeall	22b. TELEPHONE (Include Area Code): (408) 646-2509 — 22c. OFFICE SYMBOL: 52SM

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

☆ U.S. Government Printing Office: 1986—606-

Unclassified

Approved for public release; distribution is unlimited.

Petri Net and Fault Tree Analysis: Combining Two Techniques For a Software Safety Analysis on an Embedded Military Application

by

Richard J. McGraw, Jr.
Lieutenant Commander, United States Navy
B.S., United States Naval Academy, 1977

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

ABSTRACT

Software safety ensures that software programs, which control complex systems, perform within certain limits without resulting in an unacceptable risk of an accident occurring. Petri Nets are used to model concurrent computer systems by helping to understand complex interactions and paths of execution. Fault Tree analysis is used to determine safety requirements by detecting software logic errors. They also identify multiple failure sequences in a system that can lead to safety hazards. Petri Nets and Fault Tree analysis can be combined and used in conjunction with each other. They can take advantage of each others inherent strengths. This combined methodology can provide an efficient and effective safety analysis technique.

This thesis surveys software safety research and focuses on Petri Nets and Fault Tree analysis. We discuss an extended example combining Petri Nets and Fault Tree analysis. The example is a real-time, military embedded software application. We then indicate directions for further research.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I owe an enormous amount of gratitude and thanks to my advisor, Professor Timothy J. Shimeall. He has patiently and professionally guided me along the path to the completion of my thesis. Without his help, expertise, and encouragement I would not have attained the goals I set for myself. He was always available and ready to help when I required assistance. I learned a great deal of information and acquired new techniques to tackle complex problems. This will certainly help and guide me in the future and for this I am most grateful.

I am also grateful to my second reader, Major Michael L. Nelson, for his willingness to evaluate this research for accuracy and readability. I appreciate his meticulous effort.

I also would like to thank the people at the Naval Weapons Center China Lake for allowing me to tackle an operational problem. Their assistance was always timely and despite hectic production schedules you always responded to our inquires in a professional fashion. I would like to thank all the A-6E program people. Of particular note, I would like to acknowledge Werner Hueber, Scott Smith, and Bob F. Westbrook for their

supporting efforts. Their responsiveness in sending documentation was significant in aiding my progress.

Finally, I would like to thank my family. My wife Carol was instrumental in supporting my efforts during my entire stay in Monterey. She was an excellent proof reader but most of all she offered her loving support when I needed it the most. She is my best friend and I will be forever grateful.

To my children Sean, Katie, and Andrew I thank you for being so kind and understanding. Your sweet dispositions were always a cheerful reminder and a great motivator that things were really not that bad.

I. INTRODUCTION

> Computers are increasingly being used as passive (monitoring) and active (controlling) components of real-time systems, e.g., air traffic control, aerospace, aircraft, industrial plants and hospital patient monitoring systems. The problems of safety become important when these applications include systems where the consequences of failure are serious and may involve grave danger to human life and property. [Ref. 1]

Increasingly, in the world today, the trend is to implement functionality through software. Both the military and industry are becoming more and more dependent on software products. Older weapons platforms are undergoing system upgrades to significantly increase their service life that includes additional digital hardware and software. In 1955 only 10 percent of our weapons systems required computer software. Today the figure is well over 80 percent [Ref. 2].

The increase in reliance on software has led to safety critical factors that are significant. Since the techniques for software safety analysis are less mature than those of hardware safety. Much remains to be learned about applying safety considerations to the design and evaluation of computer-controlled real-time systems. Many major safety critical system purchases are now incorporating requirements for software safety analysis and verification in their contracts [Ref. 3].

The military has responded by publishing several standards for testing and verification of software systems. These include MIL-STD-SNS (1986), MIL-STD-882B (1984), and MIL-STD-1574A (1979). MIL-STD-SNS (USN) is a Navy standard that covers software safety analysis for nuclear weapons systems. MIL-STD-882B (DOD) is a Department of Defense standard for software hazard analysis and software safety verification. Finally MIL-STD-1574A (USAF) is an Air Force standard that lists the requirements for software safety analysis and integrated system safety. [Ref. 4]

Problems, however, still exist in verifying the safe states of software controlled systems. The number of states in imbedded software systems prohibit the realistic testing and verification of these systems. It is impossible to simulate all the different types of hardware errors, transient faults, and system interface errors in the design of a complex piece of software. Therefore, the overall system view is critical. The greatest source of problems encountered in computer controlled systems is in the lack of system level methods and viewpoints. [Ref. 4]

A. WHAT IS SOFTWARE SAFETY?

In order to have a common place to start some preliminary working definitions need to be presented. Complete safety cannot be achieved since nothing is completely safe under all conditions. For example, the act of

drinking too much water can cause kidney failure [Ref. 5]. Therefore, safety is a concept that needs to be measured with regards to the situation in which it is being applied.

Safety can be defined in terms of hazards or undesired states of a system. When these states are combined with other, perhaps totally unrelated, environmental factors conditions exist that could lead to a mishap. Risk is also a part of these factors. It is a function of the probability of the actual hazardous state occurring. Secondly, once the hazardous state has occurred, what is the probability of that hazard leading to a mishap? And finally, what is the perceived severity of the worst potential mishap that could result from the hazard? [Ref. 4]

Safety can then be defined as a measure of the degree of freedom from risk in any environment. Software safety, therefore, can be considered freedom from software failures that could cause damage or injury. Software, however, is not inherently unsafe. It cannot, by itself, cause physical damage or injury. The software acts in conjunction with hardware to do something that can be physically accomplished. This physical task may then have the chance to cause an incident or mishap.

Software and hardware must be treated as a single unit in order to be analyzed. Software engineering techniques that do not consider the system as

a whole, including the interactions between the hardware, software, and human operators, will have limited usefulness for real-time control software [Ref. 4].

Safety and reliability, although sometimes thought of as the same, are not. This is especially true with respect to software. Safety is the probability that a mishap or accident will not occur regardless of whether or not the intended function is performed. Reliability is defined as the probability that the system will perform its intended function for a specified period of time under a set of specified environmental conditions. [Ref. 6]

In general, for a system to be reliable it must work correctly and be failure free. The system is designed so that the software can react for every possible software error. Safety is concerned with only those errors that cause a system hazard. This is not to say that all software errors cause safety problems and all software that functions according to the specifications is safe [Ref. 3]. Serious mishaps have occurred while software was operating exactly as intended and a failure was not present [Ref. 7].

B. INTRODUCTION TO SOFTWARE SAFETY ANALYSIS

Software safety analysis needs to begin when a project is started and continued throughout the life cycle of the system. A goal of any safety analysis is to show that the system is safe. The system must be able to

operate safely under normal working conditions or the normal intended conditions. It must also be able to respond safely with the presence of errors or faults. Software or any software system must not let one simple fault disrupt its operation. The system must prove that hazards resulting from sequences of failures are also sufficiently remote. [Ref. 4]

To fully test or analyze all sequences of failures is, for all practical purposes, impossible. Therefore, the safety analysis procedures must begin to decompose the problems. An attempt must be made to define what is hazardous and then judge the likelihood of its occurrence. This approach lends itself to a backwards type of analysis. The hazards, once grouped, can then be placed in severity categories with probabilities assigned to each category. This can better define and focus which hazards would cause the worst credible mishap. [Ref. 4]

Once the preliminary hazard analysis is completed, detailed software hazard analysis can begin. The techniques in the initial stages of the analysis focus on the most damaging failures. The method of analysis starts with a given set of unacceptable failures and then by backward reasoning ensures that each failure is eliminated or at least minimized in its severity. [Ref. 4]

One method that is used to model a system is timed Petri nets. Petri nets (Peterson, 1981) allow mathematical modeling of discrete-event systems in terms of conditions and events and the relationship between them [Ref. 4].

Faults and failures can then be incorporated into the Petri net model to determine their effect on the system [Ref. 1].

Another verification methodology for safety analysis involves the use of software fault tree analysis (SFTA) [Ref. 8]. Software fault tree analysis is a static analysis of a system in order to ascertain software safety requirements. The technique can also detect software logic errors and identify multiple failure sequences involving different parts of the system. The backward approach is once again used starting from the undesired event. The sequence of actions are stepped through using gates to reach a contradiction of the undesired event. [Ref. 4]

This thesis looks at the relationship and possible integration of Petri nets and fault free analysis. Software updates to a weapons system occur quite frequently over the life cycle of a system. New and more powerful computers are added to many existing systems to increase their overall system effectiveness. These factors have contributed to more and more software being required to meet these needs. Exhaustive testing of software in many cases is not feasible. Many older systems lack formal and stringent specifications from the original system. Reverse engineering, therefore, is performed on the update to ensure that the system works correctly. If the update was limited in scope, testing can still be accomplished. However, with

today's complex and more sophisticated systems designers and implementers will not have this luxury.

Certainly, for today's safety critical systems, as much testing that can be done needs to done. However, new methodologies to analyze safety-critical computer or software controlled systems need to be developed. Combining Petri net analysis and fault tree applications may be one way to reduce the number of states or events that must be inspected. Critical states that could cause personal injury or property destruction must in some way be singled out. If these critical states are recognized and analyzed appropriate action can be taken.

Systems in many of today's applications are extremely complicated. In order to assist in a safety evaluation the system must be fully understood. Petri nets help in modeling the system in terms of conditions and events and the relationship between them. Once the system is properly understood fault tree analysis can be done in areas where possible problems exist. Today, there are no easy or widely accepted software safety techniques that have been validated completely [Ref. 4].

This thesis examines one particular real-life software upgrade. It then evaluates a method to apply Petri net and fault tree analysis to the safety issues associated with the upgrade. This analysis evaluates critical faults that

may exist in the software. This technique may help in finding and eliminating safety critical faults in other software.

The actual system under investigation is the proposed upgrade to the Operational Flight Program (OFP) 240 for the Grumman A-6E aircraft. The proposed upgrade, named OFP 250, is under development at the Naval Weapons Center in China Lake, California. Due to time constraints and levels of expertise one specific weapon system was chosen for evaluation. The actual safety analysis considered the additional functionality of the master arm weapons switch. The ability to place the master arm switch in the practice mode and interface with live weapons was not implemented in the previous release. OFP 250 changes the function of the master arm switch to let the aircrew practice and interface with live weapons carried on the aircraft. In addition, three new computers were added immediately prior to the OFP 240 upgrade. These new computers added additional complexity to the master arm practice problem.

II. BACKGROUND TO SOFTWARE SAFETY TECHNIQUES

A. SOFTWARE SAFETY DEFINITIONS

As software and computer programs have become larger the number of bugs in these programs have also grown. The ability to test for all cases in a large software program is, in practice, nearly impossible. At the same time, our reliance on computers that do critical tasks has grown tremendously. In order to deal with these inconsistencies, software safety has become a vital piece of the total system safety puzzle. Software safety analysts have agreed on certain definitions to help in defining this area of research. These definitions are set down by the IEEE Standards Committee and will be followed throughout this thesis [Ref. 9]. These definitions are listed in Appendix A.

B. SOFTWARE FAILURE MODES

When designing software some knowledge of the different types of failures that can occur should be understood. It has been fairly evident from research and by example that software does not fail due to wear. Software failures are actually program errors that exist in the system. Therefore, five system failure modes can be defined:

1. Premature operation of a component (operation not required or the operation was too early). Error of commission.

2. Failure of a component to operate at a prescribed time (operation left out of sequence or it was too late). Error of omission.

3. Failure of a component to cease operation at a prescribed time (calculation takes too long or no termination condition, infinite loop).

4. Failure of a component during operation, i.e., incorrect output.

5. Failure of a component to recognize a hazardous condition that must require some type of corrective action. [Ref. 10]

These identifiable failure modes should be the basis by which we begin to analyze key conditions in a software system. [Ref. 11]

C. SYSTEM SOFTWARE APPROACHES

Five approaches can be identified that are currently being used to enhance safety design. These are hazard elimination; hazard limitation; a group of hardware and software mechanisms that includes lockouts, lockins, and interlocks; fail-safe design; and failure minimization. A brief description of each will follow.

Hazard elimination can be done during the design phase. As the design continues to be inspected problems will be uncovered. Those design problems that lead to hazards should be eliminated. However, the problems may be dependent on hardware or the environment. A computer controlled insulin pump is an example of how the environment can effect software. The insulin

pump is set to monitor blood sugar levels in the body. A certain lower level of blood sugar corresponds to an increase in insulin. However, there are environmental factors such as diet or exercise that require different rates for the pump to add the insulin to the body. The pump must be guaranteed not to exceed a fixed maximum rate. Therefore, software cannot be the only way to eliminate a hazard. Software and hardware must work in conjunction to help solve the problem of hazards. In the above case software would need to issue commands to ensure the pump acted correctly for the given set of circumstances and blood sugar levels.

In software terms a hazard is an unsafe state. If the system is allowed to enter an unsafe state this condition can lead to a mishap or safety failure. The system was able to proceed to the unsafe state by a series of critical faults that produced a critical error. Therefore, to eliminate the unsafe state it is necessary to try and eliminate these prior errors and faults.

Some of the techniques that help eliminate hazards include formal requirements and specification languages, design tools, preliminary hazard analysis (PHA), fault tree analysis (FTA), and Petri net analysis. [Ref. 11]

Hazard limitation is another piece of the puzzle to help enhance safety design by reducing the scope of the hazard, thereby limiting the problem. The idea is to detect the hazardous condition at a low enough energy level to insure that sufficient time remains to take a recovery action.

Monitor systems must be extremely reliable. The program must be able to continually check and cross check conditions. These monitoring programs can be implemented in software. However, the software must not only monitor the situation but also take action. Warnings can be used in conjunction with the monitoring system. An aural warning for a radar altimeter set to a desired altitude or an automatic fly-up while on a terrain following mission are examples of the performance needed by monitoring software. [Ref. 11]

Lockouts, lockins, and interlocks are based on two principles:

1. isolate a hazard once it has been recognized
2. prevent incompatible events from occurring, from occurring at the wrong time, or from occurring in the wrong sequence [Ref. 10].

These mechanisms are designed to be accomplished primarily through hardware. A lockout prevents entering an unsafe state or prevents an event from actually occurring. A lockin maintains an event or keeps something from leaving a safe state. Interlocks are the mechanisms provided to avoid timing failures by ensuring that events happen in the correct order. Many of these features have been used for operating systems problems. The types of mechanisms that deal with these problems are semaphores, monitors, and kernels. Therefore, it should be safe to assume that the experience gained

from applying these techniques to operating systems will aid in safety critical applications. [Ref. 11]

Fail-Safe design tries to ensure that when a failure happens the system will remain in a safe state. The system will try to remain in or reach a state in which no damage or injury can result. Fail-Safe design can be categorized into three types: fail-passive, fail-active, and fail-operational. These fail-safe designs are described below:

1. Fail-passive design reduces a system to its lowest energy level in the event of a failure (e.g., disarming a missile after a certain period of time). A circuit breaker type fuse for the system. It should be used when no action by the system is a safe action.

2. Fail-active (fail-soft) design reduces the energy level of the other system in a stepwise fashion in the event of a failure. It should be used in systems when there is some degree of system activity that must be maintained to remain safe. Malfunctioning components are modularized. Each module can be independently terminated. This avoids a total crash of the system (e.g., water valves for cooling a nuclear power plant remain open while the system makes critical re-calculations on data).

3. Fail-operational design (fault-tolerance) ensures full system functionality in the event of a failure. It uses redundancy (parallel or switching) for critical modules that are required to maintain safety (e.g., an automatic hands off landing system for aircraft). [Ref. 11]

The last method is failure minimization. Some software systems are so vital that they cannot afford to fail. Therefore, the actual number of times the system fails must be reduced. One way to meet these goals is to try and limit failures especially under heavy load conditions. If the system is going

to fail, it is hopefully at a time when system operation can be continued in some other manner.

Redundancy is used in many cases in fail minimization systems. This does, however, become expensive. The same software cannot be used as the backup since it will have the same errors as the front line system. Different, independent software must be developed and tested to ensure that the backup works correctly. [Ref. 11]

Research continues to try and help in ways to develop methods that provide for techniques that aid in software safety. One goal is to produce safe designs for software, while at the same time, maintain system performance [Ref. 12]. Another concern is to develop ways to analyze already existing software. The design must be able to find failure modes or scenarios that could lead to a specific safety failure. Two techniques that can assist in the development and analyzing of safe software are fault tree analysis (FTA) and Petri net analysis. These two techniques will be presented for modeling of components in a system. The analysis that is done on the system will show that the information gathered can identify safety area concerns and be useful in further software safety design.

D. FAULT TREE ANALYSIS

Fault tree analysis (FTA) was developed in the early 1960's to analyze the safety of electro-mechanical systems [Ref. 13]. Evolving from FTA was software fault tree analysis (SFTA). This extended the concepts into systems that contained software as subcomponents. FTA starts by defining an undesired system state or hazard. The system is analyzed in the context of its environment and operation to find a plausible sequence of events that can lead to a hazard. The fault tree is essentially a graphic model of various parallel and sequential combinations of events or system states. The result of these combinations is the occurrence of the predefined undesired event or proof that the undesired event cannot occur.

The undesired event could occur due to component failure, human error, or environmental conditions. Therefore, the fault tree depicts the logical interrelationship of basic events that lead to an undesired event. [Ref. 14]

It is also important to understand what a fault tree is not. It is not a model of all possible system failures or reasons the system may fail. Each particular fault tree must be tailored to its singular, corresponding, failure scenario. With the possibility of an almost infinite number of ways leading to an undesired event the path chosen is the one most credibly decided on by the analyst.

E. SOFTWARE FAULT TREE ANALYSIS

Software fault tree analysis begins in a manner much like hardware fault tree analysis. The software approach uses a subset of the symbols used in hardware analysis (Appendix B). This allows software and hardware trees to be interfaced, linked, and analyzed together. It also allows for greater freedom in order to include human error and hardware failure during the analysis. [Ref. 15]

The basic procedure for FTA starts with identifying a hazard that has or could occur. The hazard is the root of the fault tree. The nodes below are the necessary preconditions that are needed to have the hazard occur. The tree builds in a backward way to determine the possible paths that may lead to this particular hazard. These relationships are built by using AND or an OR gate until each subnode has been analyzed to the lowest possible level. [Ref. 14]

After the fault tree has been built to the software interface, higher level requirements for software safety can be delineated. Unsafe software behavior may result from any number of conditions. Some examples include the failing to perform a required function, performing a function not required, failing to enforce a required sequence, or failing to recognize a hazardous condition. [Ref. 14]

Once the system has been modeled by the FTA the hazardous software behavior can be further modeled by SFTA. This can be applied at the design or code level. This analysis can lead to identifying software critical items and components, the detection of software logic errors, the determination of the initial conditions, and logical places for run time checks in the software. [Ref. 14]

SFTA is designed to work backwards just as FTA does. The SFTA attempts to verify that the program as written will not allow, under any circumstance, an unsafe state to be reached. This type of analysis does not deal with incorrect states that are not defined to be hazardous. Most of the real-time systems that are dealt with have two goals in mind. One is accomplishing the mission or function at hand. The second is to not cause harm, while in the process of accomplishing the mission. SFTA only addresses the second portion of stated the goals. [Ref. 14]

Proof by contradiction is conveniently used in SFTA since the goal of the analysis is to prove that the software will not permit some event. If the analysis can prove a contradiction to the loss event then the event cannot happen with the software. SFTA forces the analyst to consider what the software is not supposed to do. This thought process is opposite the normal approach of what is the system software required to do. The environment is

also considered by this approach. Therefore, the most critical assumptions about the environment can also be considered. [Ref. 14]

F. PETRI NET ANALYSIS

Petri net theory was originally developed by A. W. Holt and others based on the works of Carl Adam Petri [Ref. 16]. Petri's efforts were directed to design a new model of information flow in a system. As computer systems have developed, concurrency of operations have become more and more common. This has lead to complex interactions between concurrently executing components. This makes it nearly impossible to understand the entire system. [Ref. 17]

Petri nets have been developed to aid in the understanding of concurrent systems because they have the ability to model parallelism and synchronization. This new approach realized that relationships between components of a system could be represented by a graph or net. [Ref. 17]

The analysis by a Petri net can uncover possible problems within the system and get them corrected. The ultimate goal would be the ability to analyze a system using a Petri net and then manipulate the net to derive the properties of the modeled system. Questions can be raised that significantly enhance the ability to investigate the problem. For example, what states are

reachable in a given Petri net? What different sequence of transition firings are possible? [Ref. 17]

Petri nets are defined in computer science terms as directed graphs whose nodes are transitions and places. Places are connected to transitions and transitions are connected to places by directed arcs. Places model conditions and transitions model the occurrence of an event. Inputs or arcs leading into a transition represent the precondition of the event. The arcs leading from a transition define its outputs or postconditions. Figure 2-1 represents the basic Petri net structure.

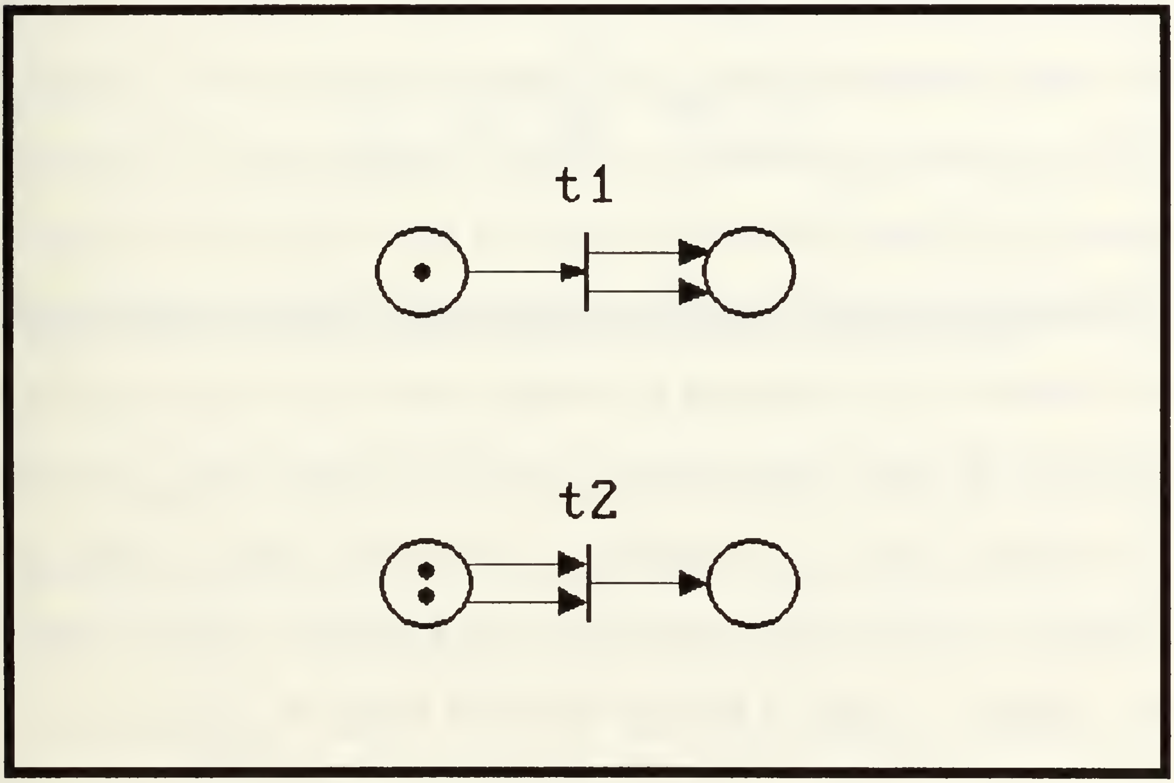


Figure 2-1 Basic Petri Net Structure

The circles of Figure 2-1 denote places, the bars are representative of transitions, the dots represent tokens, and the arrows represent the arcs of the graph.

Petri net graphs represent the static state or condition of the system under analysis. In addition, Petri nets can also be used to analyze the dynamic properties that result from execution. Each place can contain tokens that model the dynamic properties of the system. Tokens are indicated by black dots inside of the places as in Figure 2-1. A transition cannot fire if all the tokens are not present. This then allows for simulation of the model so that it can be analyzed under different sets of conditions. Tokens are positioned and moved throughout the net by the firing of transitions. The transitions must be enabled in order to fire. A transition is enabled when all of the input places have a token in them. The number of arcs coming out of a transition represents the number of tokens that will be created when the transition fires. The transition fires by removing the enabling token from the input places and depositing the newly created token or tokens into the output places. It should be noted that there is no dependency between the number of input arcs required to enable a given transition and that transition's number of output arcs. Figure 2-1 depicts a basic net with input tokens set to fire.

Figure 2-2 shows what has occurred after the transitions in Figure 2-1 have fired. The token is placed into the next place corresponding to the number of input arcs that lead in.

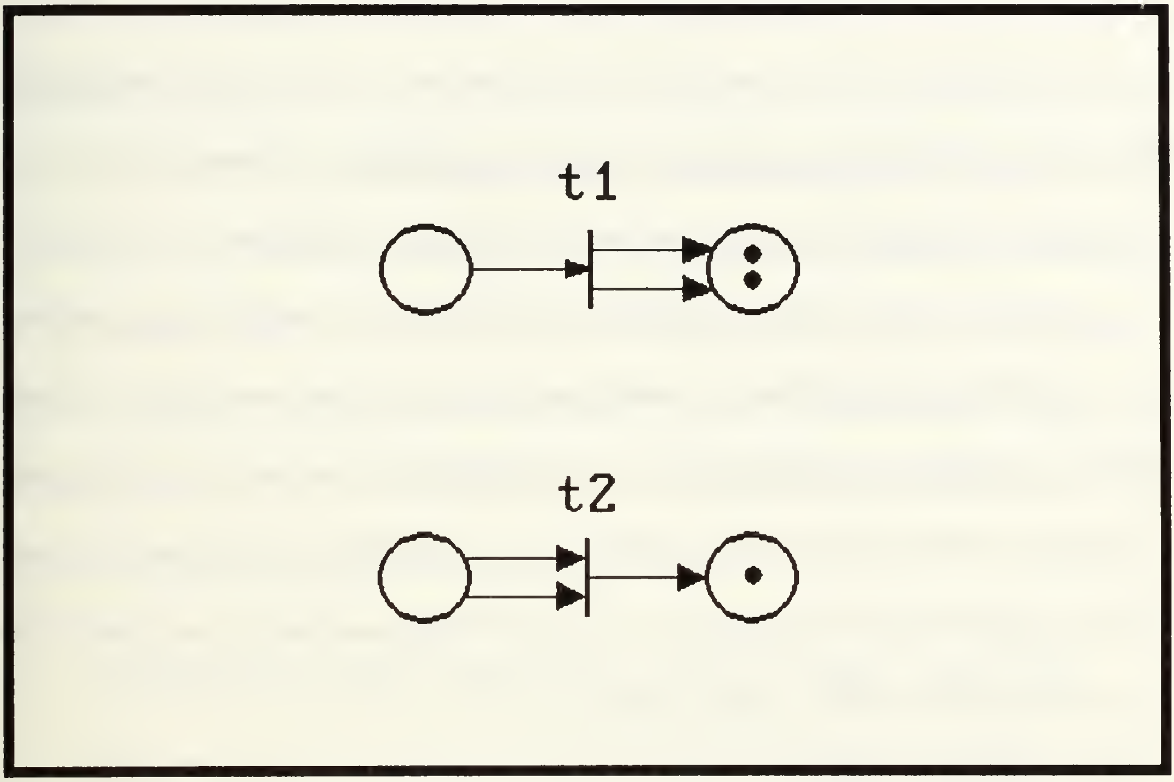


Figure 2-2 Basic Petri Net Structures After Transitions Firing

A Petri net execution can be viewed as a series of discrete events that can occur depending on the basic structure of the model being analyzed. This is an important concept since this leads to the idea that Petri nets can fire in a nondeterministic manner. If more then one transition is enabled the possibility exists that any one of those transitions may fire, with the one that

fires chosen nondeterministicly. The selection process can be purely random or driven by forces that are not included in the model.

The nondeterministic actions of Petri nets matches well with events in real life. This is ideal since real-life events occur in no specific set pattern. Each occurrence can be a unique set of sequences. It is this fact that makes Petri nets suitable for concurrent system modeling. However, this does introduce considerable complexity into the analysis of the net.

A way to reduce the complexity of the Petri net model is to consider that the firing of transitions is instantaneous or takes no time to fire. Since, time is a continuous variable, then, the probability of any two or more events happening simultaneously is zero [Ref. 17]. One other way to remove some of the nondeterministic behavior is to set up timing maximums and minimums for the transitions firing times. [Ref. 17]

1. Petri Net Theory

The formal definitions of Petri nets can be found in Peterson (1981). A Petri net is a five tuple relationship. It consists of a set of places P, a set of transitions T, an input function I, an output function O, and an initial marking σ. [Ref. 17]

2. Reachability

If there is a possible transition between one place to another then we can say that the marking is immediately reachable. In other terms,

reachability is the possibility that an initial condition could lead to a given final condition. Reachability is a directed graphical representation of all the possible state sequences. The nodes represent states. The arcs between the nodes represent sets of transitions which sequentially go from one state to another.

Petri net safety analysis find this advantageous because it uses reachability to determine if there is a possibility of a mishap state occurring. If a reachable unsafe state is discovered it can then be analyzed. The analyst must determine where a critical state is first encountered. With this information the error can be corrected by changing those conditions that led to that specific critical high risk state. [Ref. 17]

G. COMBINING PETRI NETS AND FAULT TREE ANALYSIS

Petri nets and fault tree analysis (FTA) have been recently used in conjunction with one another by Leveson and Stolzy, 1986 [Ref. 1]. They have developed analysis procedures to help in determining software safety requirements directly from the system under investigation. They have also included timing requirements, recoverability, and fault tolerance to help in determining failure detection and recovery procedures.

To generate the entire reachability graph from Petri nets has been shown to be exponentially difficult and time consuming. However, by using Petri

nets for only key parts of a system the same type of backward analysis that is used in FTA can be used in the Petri net. The state information provided by a fault tree can also be used to reduce the exponential nature of the Petri net analysis.

Timing can also be added to the Petri net model. This approach can help in determining the longest time required or worst case scenario required to complete execution. This information can then be incorporated into the design currently under development.

The Petri net incorporates faults and failures into the model. The backward analysis helps uncover problem areas where critical hazards exist. The analyst can then take the most hazardous part of the system and correct it by using fault-tolerance and fail-safe mechanisms. If the hazard is too severe it may need to be eliminated. [Ref. 1]

The above approach attempts to establish important properties of the system through a framework of examination instead of guesswork. It uses the advantages of timed Petri nets and the ability to model the flow of execution with likely fault tree failure scenarios. It may be particularly useful for software components. Since it is difficult to determine in a complex system which faults are the most likely to occur and the number of failures is often very large. [Ref. 1]

III. MODELING AND ANALYSIS METHODOLOGY

A. INTRODUCTION

The majority of fleet training with air-to-ground missiles is conducted with live weapons. The real-time systems employed on different platforms throughout the Navy have required thorough safety inspections and analysis. The safety analysis of our system used the modeling power of Petri nets to trace the flow of events as the system executed. This was critical to analyzing the system considering that four computers were involved which were working concurrently. Three of the computers were recent upgrades to the aircraft. Relatively new interfaces were modeled and examined to determine if any new hazardous conditions existed. With this increased knowledge of the flow of information the unsafe states were identified. FTA was then used to analyze if the unsafe states could possibly harm the system.

B. SYSTEM OPERATION

The real-time system under analysis is the upgrade of the A-6E operational flight program (OFP 240). The improvements to the software are a continuing program controlled and directed by NWC China Lake. The upgrade, called OFP 250, will accomplish several tasks. It will combine

OFP 230, used for older A-6E versions, and OFP 240 which includes the 1553 data bus and new computer hardware. OFP 250 will also add the capability to perform missile practice attacks with the aircraft master arm switch in the practice position.

Live missiles will be required to interface with the system. The practice attack procedures will be identical to those required for a real missile attack. The practice attack software will also encompass all current A-6E air-to-ground missiles.

Our analysis of the system was conducted on OFP 240. The evaluation examined the potential condition of an inadvertent release of a missile with the master arm set to practice and a live weapon aboard the aircraft.

The A-6E, before the upgrade, used one central computer called the 4PI or Ballistic Computer Set (BCS) to control the entire system. The new configuration added three new computers along with the 1553 data bus. The central or key component of the new system is the Avionics Interface Unit (AIU). [Ref. 18]

In addition to the AIU there are two other computers that need to be considered to understand some of the concurrent operations. These two computers are the Missile Switching Unit (MSU) and the Integrated Missile Panel (IMP). A system block diagram is provided in Figure 3-1 [Ref. 18].

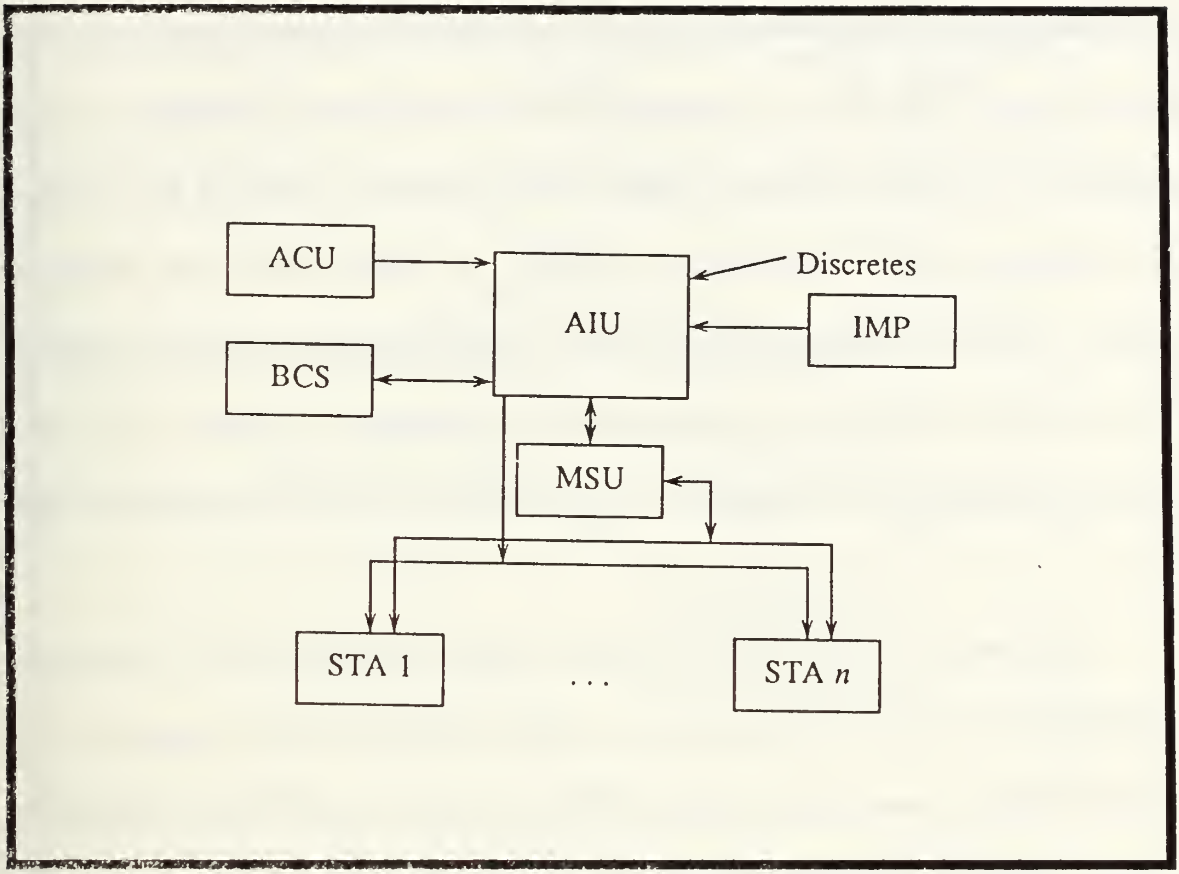


Figure 3-1 System Block Diagram

The concurrent operations of all the individual system components is very complex. We therefore started our analysis by using a timing diagram to study the sequential flow of execution for a live air-to-ground missile launch.

The specific missile that we considered was the Harpoon. Harpoon was decided upon for a number of reasons. It was a proven weapon on the A-6E airframe and is representative of other air launched missiles. It was also the area in which most of our expertise and operator knowledge existed.

Using the timing diagram and a schematic of the signals for the input/output of the AIU, a simplified flow chart diagram was made. This enabled us to better understand some of the concurrent events in the system. The timing sequence was used to identify the signals from the Harpoon missile. The AIU diagram was then used to help match and map the signals that interfaced between the Harpoon and the hardware. Figure 3-2 is the resulting simplified block diagram of the interface between the Harpoon and AIU.

The Harpoon weapon was somewhat unique considering that it had been implemented in older A-6E aircraft by using a 100-kHZ data channel. The 100-kHZ data channel is still used with the 1553 data bus in new or upgraded versions of the aircraft. The analysis was forced to take this 100-kHZ bus into account in examining safety-critical events. It did not, however, effect our results. [Ref. 18]

C. PROBLEM BEING ANALYZED

Prior versions of OFP 240 did have some practice missile capabilities. These capabilities, however, were restricted to the left and right outboard stations. These stations are unable to be used by any type of weapon, either live or practice. The circuitry to the outboard stations exists, but there is no physical hardware that allows the weapon to be attached to the aircraft at

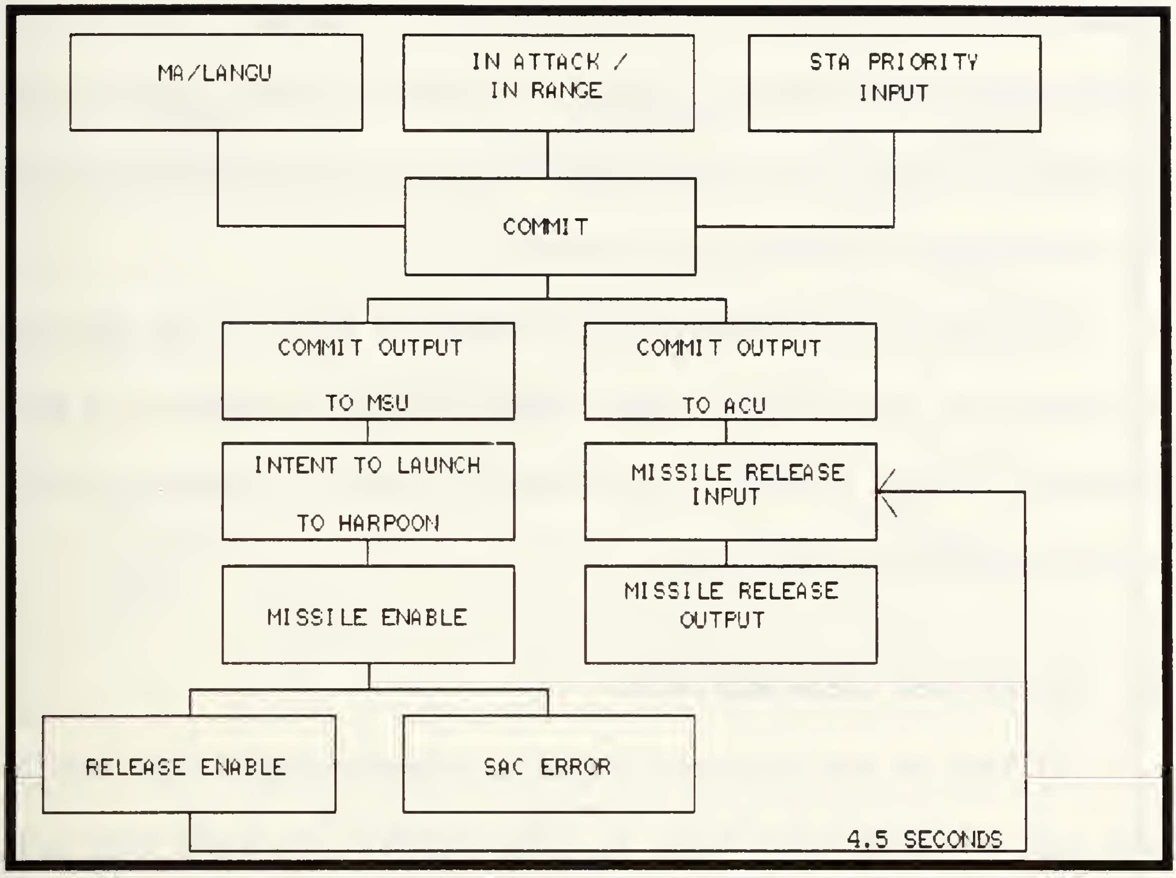


Figure 3-2 Simplified Block Diagram

these stations. This limitation seriously degraded real-life training scenarios.

The ability of the system to interface between live weapons and the computer for practice attacks had not been attempted previously. Realistic training prior to this upgrade could only be conducted by placing the master arm switch to on. This was not acceptable since this action caused the pilots release mechanism to be armed and functioning.

In order to minimize the potential for an accident with a live weapon the master arm switch must be positioned to practice. If this is the case then the

system must be allowed to interface with the live missile. The practice missile attack will, therefore, require the aircrew to perform all the steps necessary to launch a real missile. The difference being that the missile will not receive the actual fire pulse. [Ref. 18]

The portion of the problem that we began to focus on was after the communication link between a live Harpoon and the computer had been established. Could a missile inadvertently fire with the master arm set to practice during the practice attack?

D. PETRI NET DESCRIPTION

The Petri net was developed to gain an understanding of what went on with each individual signal during the firing sequence. Two nets were built, one for a live missile fire and one for a practice missile fire. Our goal was to examine the functionality of the nets to ensure that they behaved in the same manner as our actual knowledge of the system.

The live fire net is represented in Figure 3-3. The area of main interest centered around the commit high signal. There are several prior preconditions that occur before the live fire can be initiated. Three of these signals occur in both of the nets. These signals are the internal priority, station priority output high, and the pilot control stick (PCS) input. The dashed lines in Figure 3-3 represent command abort conditions.

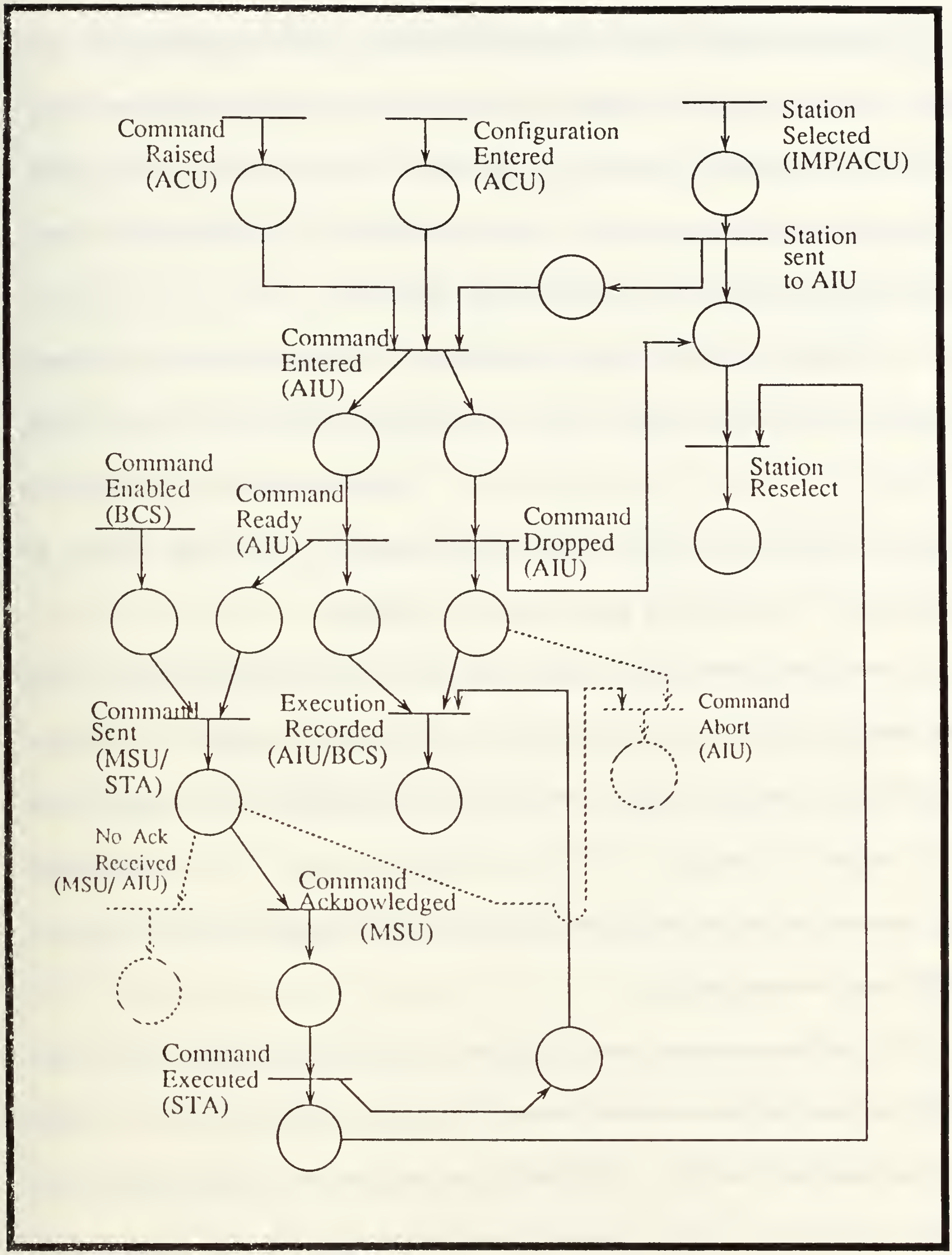


Figure 3-3 Live Fire Net

The input of the missile practice mode being false comes from the IMP. This is a safety check to ensure that the operator has entered the correct type of attack, either live or practice, in the IMP. This check must correspond to the position of the master arm switch located on the armament control unit (ACU) and the entered type of attack in the IMP.

The master arm/landing gear up (MA/LANGU) input is the last and most important signal that appears in the preconditions for the live fire net. This signal does not occur in the practice net. It must be set true or high and remain high throughout the entire firing sequence. If for any reason the signal goes low the firing sequence will be aborted.

Once the commit high signal from the PCS goes high the AIU begins the launch sequence. The AIU responds by generating signals to the MSU and ACU. These two signals are produced concurrently. The signal to the MSU begins the interface with the actual Harpoon missile. This branch of the net does not occur in the practice fire net and is a significant and important difference between the two.

The other portion of the fire signal is sent from the AIU to the ACU. When all preconditions are met, the ACU generates a firing pulse. This pulse is sent back to the AIU. The AIU then transfers the firing pulse to the pylons by way of a station missile release output signal and the weapon fires.

The practice fire net is presented in Figure 3-4. It is somewhat simpler considering that it does not rely on the MA/LANGU input. The initial difference is a new input is introduced from the BCS labelled master arm practice (MA PRAC). This signal is a new and separate signal to begin the sequence for a practice attack with live weapons. The MA PRAC input also has the same safety function implementation as in the IMP design from the live fire net. The IMP and master arm switch position on the ACU must be in agreement to work properly.

Once the commit high preconditions have been met the practice fire net produces one signal versus the two found in the live fire net. The practice net does not enter the path to the MSU and the subsequent interface between the MSU and Harpoon. The Harpoon is thereby isolated by the practice fire procedures and no signals are received by the missile during the critical firing sequence.

The system continues and as the preconditions of the net are met the AIU outputs the commit output high to the ACU. It will also output commit true to the IMP and set the internal function commit conditions to true. Finally, it will monitor the system for the ACU release signal from the BCS contingent on a good or bad launch. The BCS then provides all the realistic release parameters and signals to all other system components as if it were a real launch. This would include most error codes or abort procedures that

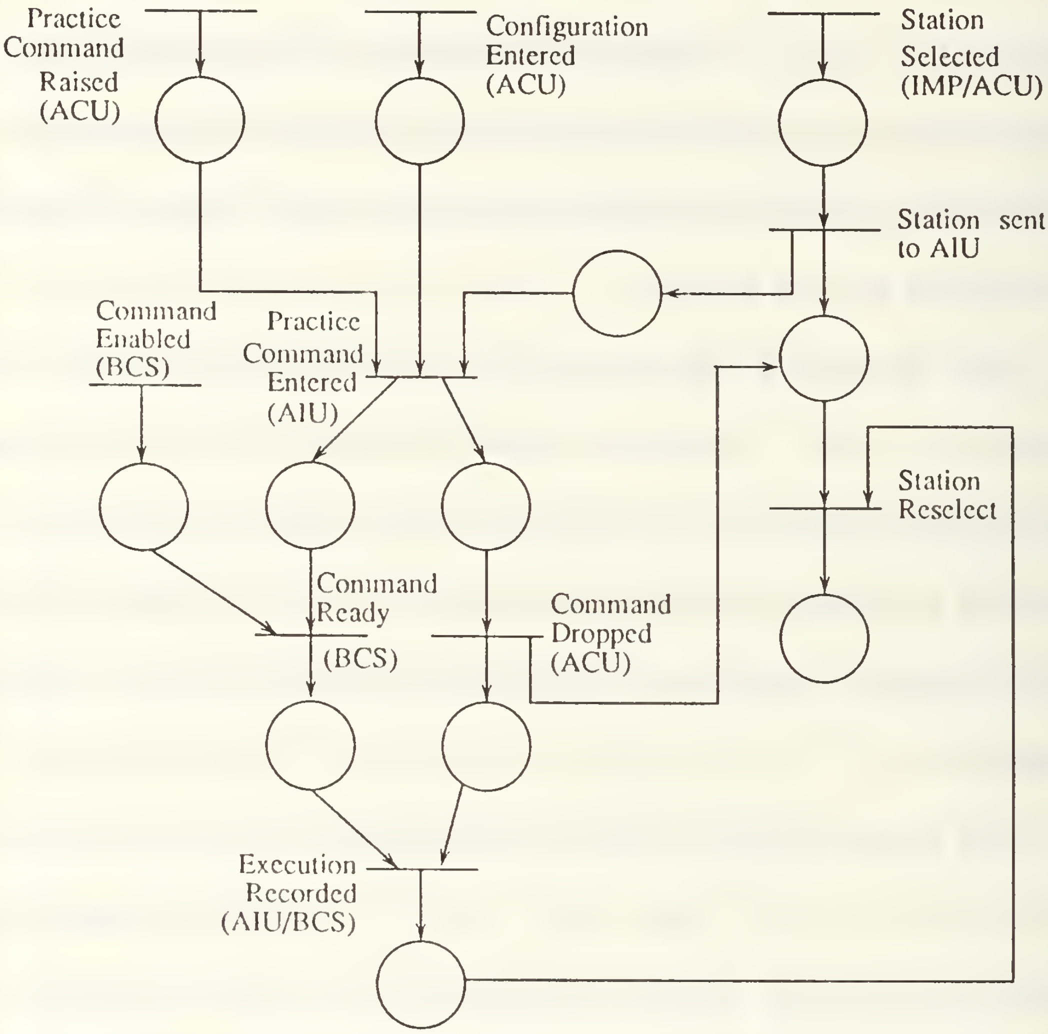


Figure 3-4 Practice Fire Net

may occur during the release of the weapon. The MA/LANGU procedures would not be included because of their absence in the practice net.

E. FAULT TREE ANALYSIS

Building on the knowledge obtained from the Petri net modeling the fault tree analysis could begin. With the Petri net modeled and organized certain information was discovered to be vital. The knowledge of certain preconditions and paths in the net helped in assessing the fault tree analysis.

The MA/LANGU was a condition that only existed in the live fire case. The live fire case also generated two signals after commit, one of which actually interfaces with the Harpoon.

The practice fire case had a new input from the BCS that was MA/PRAC. This was critical to the practice event occurring and also caused the practice fire Petri net to be different than the live fire Petri net. The practice fire case did not interface with the Harpoon during the firing sequence.

The overall problem areas in the system can be reduced because of the knowledge obtained from the Petri nets. In our specific case two scenarios were studied. The first was the case of a Harpoon firing with the master arm in practice with the commit occurring normally. The second condition or case

occurred if a Harpoon fired with the master arm in practice with no commit present.

The first case is presented in Figure 3-5. The Harpoon fires with the master arm in practice. Discounting that a short circuit occurred in the master arm switch, two paths are explored. First to the left, the fire signals are sent from the AIU to the pylon. If this is the case, station missile release signals would have to be high and this could only occur if the master arm is in the on position. This is a contradiction to the initial loss event and therefore could not happen.

The track of the MA/LANGU also continues down the tree and comes to the master arm contradiction in each of its paths. The contradiction event can be reached after the system has sent MA/LANGU to the AIU. If the MA/LANGU signal is accurate, then the master arm must be on. This is not the case and therefore causes a contradiction.

Following the right hand path of the tree the first condition reached is that the missile is ready to fire. The ready to fire condition is followed by the MSL enable signal. Finally, if the signal to the MSU is present, the precondition for the event is that MA/LANGU must be true. The master arm is in practice and therefore a contradiction is reached.

The second scenario that was discussed was the occurrence of a missile firing without a commit present and the master arm in practice. The fault tree

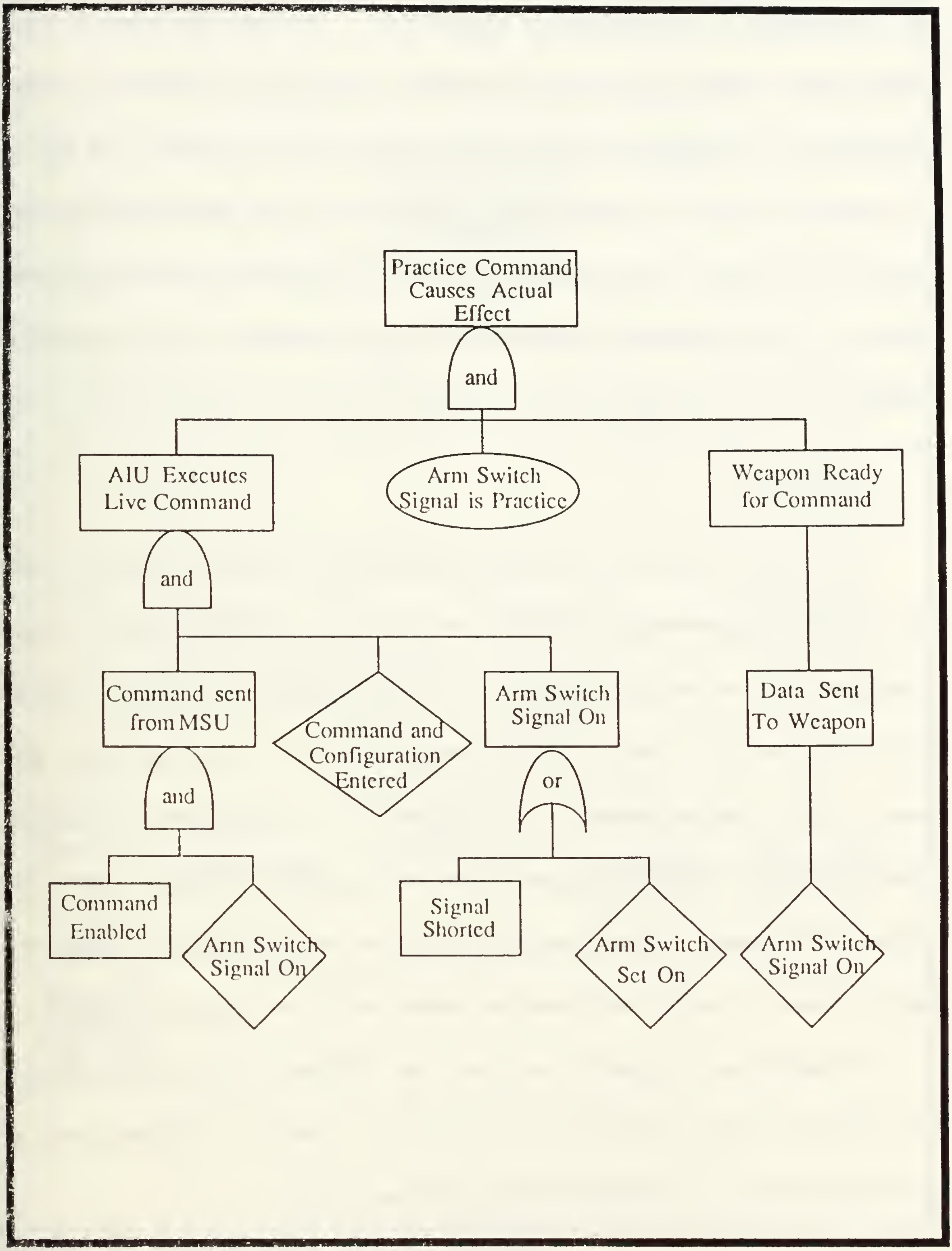


Figure 3-5 Missile Fires With Commit

for this scenario is represented in Figure 3-6. This tree also goes through station missile release signals from the AIU to the pylon. However, in order for this event to happen the commit high signal must be present. If this is the case then the PCS input must be present from the pilots stick for the commit to go high. The commit high signal for the initial loss event is not present. A contradiction is identified and the analysis need proceed no further.

F. SUMMARY

The analysis from the two scenarios shows that a Harpoon missile could not be fired inadvertently. There were enough safeguards and design techniques that ensured that a path to a live fire missile could not be reached in a practice attack. Two cases of an analysis, however, does not mean that there are not other problems areas that need to be investigated. Different missiles may have different signals that are required. However, with the addition of a separate output from the BCS to the AIU for practice attacks the design appears to enable safe practice attacks with live Harpoon missiles.

The additional safeguards built into the IMP and ACU switch settings also help ensure safe operations of the system. These are excellent ways to go about insuring an inadvertent firing will not occur.

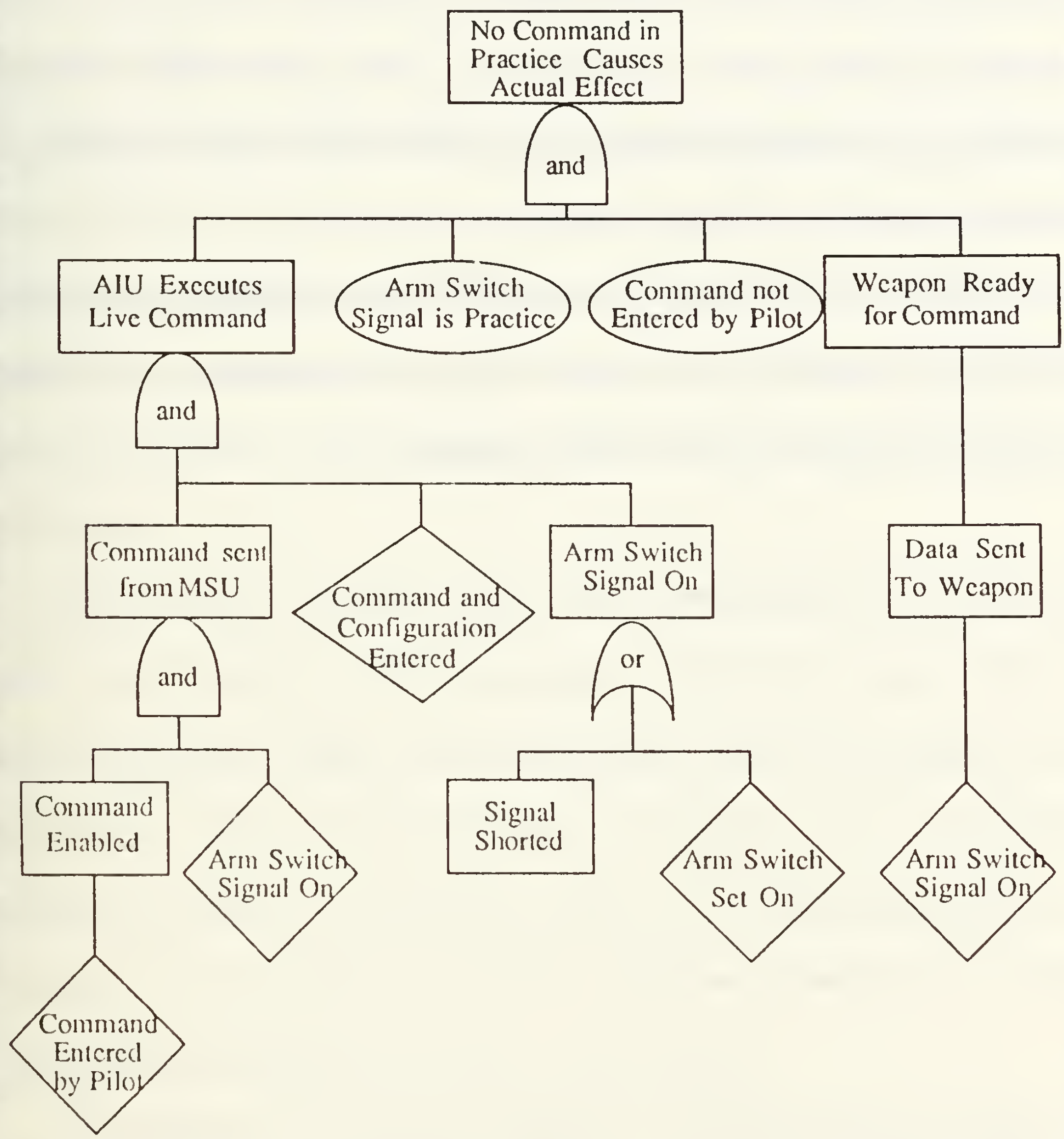


Figure 3-6 Missile Fires Without Commit

There are cases and scenarios in any system that can not be identified or thought of at the time of the design and implementation. Therefore, a methodology of putting the system through an analysis of some type to reduce the amount of unsafe states is advantageous. There will be failure scenarios that need to be continually identified and analyzed by experts to ensure the system continues to meet safety criteria.

In examining the two specific cases we looked at the inadvertent release and accidental possibilities of a Harpoon firing. The interlocks in the design of the system block these events from occurring.

IV. RESULTS AND CONCLUSIONS

A. REVIEW

The software safety analysis of a concurrent system in a multiple and distinct CPU environment is a complex one. This problem is certain to become more complex and complicated in the near future. Systems under consideration, such as the A-12 aircraft, the Advanced Tactical Fighter (ATF), and some portion of space defense will have large numbers of interacting subsystems.

These systems will also control and be responsible for more and more safety critical functions. The environment in which they operate will also be more demanding. Design faults and problems must be determined at earlier points in the software development cycle. Increasing costs and complexities are just two reasons answers must be found. New methodologies need to be discovered and perfected to insure that sophisticated hardware and software does not cause catastrophic incidents and accidents.

This thesis has investigated (proposed) a method for using the power of Petri nets to model concurrent systems in a multiple CPU environment. The information obtained from the modeling of the net will help reduce the number of cases that need to be analyzed. With the knowledge of the system

well understood examining the reduced set of loss events can occur with fault tree analysis. FTA can then look at each individual loss event and decide on the feasibility of the event actually occurring.

The sample system we chose is a proposed upgrade to an already existing operational flight program (OFP 240) for the A-6E Grumman aircraft. The modeling and safety analysis for this system examined a real-time system currently in the United States Navy inventory.

The analysis initially used Petri net modeling to begin to break down and organize complex interactions of the system. Using the methodologies of Petri nets all aspects of the system functionality were analyzed, including the different internal interfaces to other computers. The analysis then moved into a constructed block diagram to allow for an examination of individual components and a thorough study of component operation, control flow, and system interfaces.

The model centered around the AIU that is the heart of the new system. The Petri net modeling technique was used to understand the concurrent aspects of the AIU with the Harpoon air-to-ground weapon. The Harpoon was used since it was representative of other weapons that could interface with the AIU.

Key signals and information routes were determined after the live fire and practice fire nets were modeled. Differences between the two nets were

identified. This gave the analyst a sense of where problems could occur in the firing sequence.

Once the live and practice fire nets were understood the analysis moved into FTA. The nets allow the analyst to ascertain the loss event scenarios that required the most attention. The analysis of these loss events, however, is time consuming. Automatic tools are being developed to assist in this area of analysis and need to be developed further.

In our scenario two loss events were analyzed. The analysis begins by describing the loss event and follows the standard technique of FTA. The loss event in each case was the root of the tree. The events or nodes below were the necessary preconditions that were needed for the hazard to occur. It was shown in each case that there were many places in which a contradiction was identified. Therefore, it was determined that neither loss event could occur. There were sufficient safeguards in the design that prohibited the inadvertent release of a Harpoon in a practice attack.

B. RECOMMENDATIONS

We have demonstrated the feasibility of applying Petri net modeling to a complex concurrent problem of software safety analysis. We then used this information to create specific fault tree applications. We did not design a formal model of these conditions.

The methodologies presented are only a preliminary step in creating a complete set of rules to identify when Petri nets and fault tree analysis can be used. Leveson and Stolzy, for example, have used a methodology of simulating system faults within a Petri net model. Their techniques added fault transitions to the net to cause unintended events or prevented intended events from occurring [Ref. 1].

Petri nets are excellent for modeling the concurrent actions of a system. Petri nets are also strongly suited for timing constraints. Systems that proceed in parallel and need real-time synchronization benefit from Petri net modeling. FTA can then be used for logical event analysis and specific problem solving techniques.

Petri nets, however, are difficult and time consuming to analyze. In order to analyze a complex concurrent system automatic tools are needed. The reachability graphs must be generated automatically by tools. Any valid technique to reduce the time and enormous number of paths in a problem will be essential for the future.

In a complex software system consideration needs to be given to which techniques will help provide the best possible support for analyzing the system. Problems will occur during the design of a complex system. Most of these problems will be detected and corrected. There must, however, be techniques to uncover hidden problems. The application of Petri nets and

fault tree analysis techniques can make a difference in building and designing better systems.

In a system with a complex set of concurrent operations, Petri net methods should be used first to drive the fault tree analysis. The concurrent operations are better suited to be analyzed by Petri net models initially. Once the complex system is understood problem areas can be better analyzed. FTA can then be used where experts think major problems are likely to occur.

In a situation or system where events in a fault tree depend on specific concurrent states being reached, fault tree methods should drive the analysis. Fault tree analysis is better suited for analyzing specific events. Petri nets can be then used to model the particular concurrent events in the actual tree. Deciding on which approach to use is a function of the system under analysis.

One of the major concerns with Petri net methods is the difficulty in constructing graphs to model the actual system. This is a complex task and is an area that is currently undergoing major research initiatives.

Integrated tools are one approach that is being investigated to reduce the time and complexities of Petri net analysis. We recommend that this thrust should also include tools that support not only Petri nets or fault tree analysis but both. The information that is obtained from these two techniques is certainly vital to overall system performance.

These tools, once designed, should be integrated. The information obtained in a Petri net model may also be able to be used by a fault tree analysis. Questions arise with regards to what portions of the information in the analysis are common to both techniques. Work in the field of Petri nets and FTA must be done on the inter-relationships between these two techniques. Petri nets are versatile enough to enable accurate modeling of many concurrent system aspects. They capture the features of system interfaces and paths of execution and are dynamic assets.

Fault trees are essentially a static analysis. They can, however, detect software logic errors and multiple failure sequences that may have essential information that can be shared with Petri net analysis.

Other concurrent modeling techniques also need to be explored. One example is communicating finite state machines with shared variables, a technique that allows for direct representation of race conditions. Race conditions, where two processes write to the same location simultaneously, have been shown to contribute to some past mishaps.

We have introduced a technique to combine Petri nets and fault tree analysis. These combined techniques have the possibility to help in determining the future of safety analysis. Answers need to be found to ensure that critical software components can be judged to be safe. Software errors, including simple oversights, must not be the cause of accidents or incidents

once the software is introduced for public use. Therefore, we strongly encourage further research in this area and other areas introduced in this thesis.

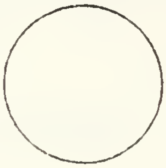
APPENDIX A

Failure -	The inability of a system or system component to perform a required function within specified limits. A software failure occurs when the failure is due to a software fault.
Fail-Safe System -	A system which limits the amount of damage caused by a fault. No attempt is made to satisfy the functional specifications except where necessary to ensure safety.
Hazard -	A condition with the potential for causing loss of life or property.
Loss Event -	A hazard at the top level of a fault tree that, if the event occurred, could cause a mishap.
Reliability -	The probability that a system, including all hardware and software subsystems, will perform a required task or mission for a specified time in a specified environment.
Safety -	The ability of the system to avoid safety failures.
Safety Failure/Mishap-	A failure which leads to casualties or serious consequences. A serious consequence is any undesired event which the designer considers to be as or more important than the correct (reliable) operation of the system.
Safe System -	One which prevents unsafe states from causing safety failures.
Software Error -	A human action or inaction (during development or maintenance) which results in software containing a fault.

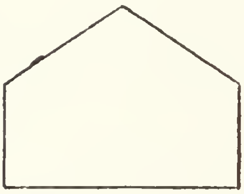
Software Fault -	A manifestation of an error in software. A fault, if encountered, may cause a failure.
Software Safety -	The ability of the software system to avoid safety failures caused by software errors.
Unsafe State -	A state from which there are circumstances where further processing will lead to a safety failure or hazard.

APPENDIX B

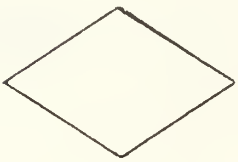
The *rectangle* indicates an event to be analyzed further.



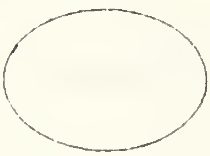
The *circle* indicates a basic fault event or primary failure of a component. It requires no further development, and its probability of occurrence is derived from the generic rate of the part.



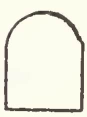
The *house* is used for events which normally occur in the system. It represents the continued operation of the component, and its probability is the reliability of the part.



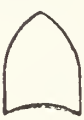
The *diamond* is used for non-primal events which are not developed further for lack of information or insufficient consequence.



The *oval* is used to indicate a condition. It defines the state of the system that permits a fault sequence to occur. It may be normal or result from failures.



The *AND* gate serves to indicate that all input events are required in order to cause the output event.



The *OR* gate indicates that one or more of the input events are required to produce the gated events.

LIST OF REFERENCES

1. Leveson, N. G., and Stolzy, J. L., "Safety Analysis Using Petri Nets", *IEEE Transactions on Software Engineering*, vol. SE-13, no. 3, March 1987.

2. Griggs, J. G., "A Method of Software Safety Analysis", *Proceedings of the Safety Conference* (Denver, CO), vol. 1, part 1, System Safety Soc., Newport Beach, CA, pp. III-D-1 to III-D-18, 1981.

3. Ericson, C. A., "Software and System Safety", *Proceedings of the 5th International System Safety Conference* (Denver, CO), vol. 1, part 1, System Safety Society, Newport Beach, CA, pp. III-B-1 to III-B-1, 1981.

4. Leveson, N. G., "Software Safety: Why, What, and How", *Computing Surveys*, vol. 18 no. 2, (June 1986), pp. 125-163, 1986.

5. Gloss, D. S., and Wardle, M. G., *Introduction to Safety Engineering*, Wiley, NY, 1984.

6. Konakovsky, R., "Safety Evaluation of Computer Hardware and Software", *Proceedings of COMPSAC '78*, IEEE, NY, pp. 559-564, 1978.

7. Roland, H. E., and Moriarity, B., *System Safety Engineering and Management*, Wiley, NY, 1983.

8. Leveson, N. G., and Harvey, P. R., "Analyzing Software Safety", *IEEE Transaction Software Engineering*, SE-9, 5(September), pp. 569-579, 1983.

9. "Computer Dictionary", *IEEE Computer Society Standards Committee*, Martin Weik, ed., IEEE Computer Society, 1979.

10. Hammer, W., *Handbook of System and Product Safety*, Prentice-Hall, Inc., 1972.

11. University of California, Irvine, Computer Science Technical Report, "Applying Existing Safety Design Techniques to Software Safety", by Jeffery C. Thomas, and Nancy G. Leveson, September, 1981.

12. Harvey, Peter Randll, "Fault Tree Analysis of Software", M. S. Thesis, University of California, Irvine, CA, 1982.

13. Vesely, W. E., Goldberg, F. F., Roberts, N. H., and Haasl, D. F., *Fault Tree Handbook*, NURREG-0492, U. S. Nuclear Regulatory Commission, January, 1981.

14. Cha, Stephen S., Leveson Nancy G., and Shimeall, Timothy J., "Fault Tree Analysis Applied to Ada," *Proceedings of the Tenth International Conference on Software Engineering*, Singapore, 1988.

15. *Software Safety Handbook (Draft)*, H.Q. AFISC/SESD, Norton Air Force Base, CA, March, 1984.

16. Petri, C., *Kommunikation mit Automaten*, Ph.D. dissertation, University of Bonn, Bonn, West Germany, 1962.

17. Peterson, J. L., *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

18. MIL-STD-1553B (DOD), *Specification Control, Missile C-11524/A (AIU), Program Performance Specification (PPS) for A-250, NWC-2478-250 Revision C*, Naval Weapons Center, China Lake, CA, 1 June 1989.

BIBLIOGRAPHY

Connolly, Brain, "Software Safety Goal Verification Using Fault Tree Techniques: A Critically Ill Patient Monitor Example", *Proceedings of IEEE Compass '89*, Gaithersburg, MD, pp. 18-29, 19-23 June 1989.

Leveson, N. G., and Stolzy, J. L., "Using Fault Trees to Find Design Errors in Real Time Software", *AIAA 21st Aerospace Science Meeting*, Reno, NV, January 10-13, 1983.

McKinlay, Archibald, "Software Safety Handbook", *Proceedings of IEEE Compass '89*, Gaithersburg, MD, pp. 14-19, 19-23 June 1989.

Neumann, Peter G., "The Computer-Related Risk of the Year: Misplaced Trust in Computer Systems", *Proceedings of IEEE Compass '89*, Gaithersburg, MD, pp. 9-13, 19-23 June 1989.

A-6E Operational Flight Program E 250, Mini-PPS, Revision B, Naval Weapons Center, China Lake, CA, 93555, 23 June 1989.

A-6E 4 PI Developmental Flight Program, E 544.06, Math Flows Draft, Naval Weapons Center, China Lake, CA, 93555, 17 May 1989.

MIL-STD-1553B (DOD), *Specification Indicator, Display/Control ID-2369/A (IMP), Program Performance Specification (PPS) for I-250*, Naval Weapons Center, China Lake, CA, 93555, 1 June 1989.

MIL-STD-1679 (DOD), *Control Missile C-11524/A (AIU) and Ballistics Computer Set*, CP-1391/ASQ-155A (BCS), Interface Design Specification (IDS), NWC-2482-250, Revision A, Naval Weapons Center, China Lake, CA, 93555, 1 August 1989.

University of California Irvine, Computer Science Technical Report NO. 86-14, *Building Safe Software*, by Nancy G. Leveson, February, 1986.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	1
4.	Professor Timothy J. Shimeall, Code 52Sm Computer Science Department Naval Postgraduate School Monterey, California 93943-5000	10
5.	Major Michael L. Nelson, USAF, Code 52Ne Computer Science Department Naval Postgraduate School Monterey, California 93943-5000	2
6.	Mr. Bob F. Westbrook (Code 31) Naval Weapons Center China Lake, California 93555	2
7.	Mr. Werner Hueber (Code 3104) Naval Weapons Center China Lake, California 93555	3
8.	LCDR Richard J. McGraw, Jr., USN 165 G Ave Coronado, California 92118	8